Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 65 (2007) 4–13
Science of Computer Programming
www.elsevier.com/locate/scico

What business process modelers can learn from programmers

Volker Gruhn*, Ralf Laue

Computer Science Faculty, University of Leipzig, Klostergasse 3, 04109 Leipzig, Germany[1]

Received 15 April 2006; received in revised form 1 July 2006; accepted 1 August 2006
Available online 18 October 2006

Abstract

For building business process models (BPM), business process analysts usually use graphical languages like BPMN or UML. One purpose of such models is to serve as a base for communication between the stakeholders in the software development process. Furthermore, modern model-centric software engineering approaches have the potential to enable the generation of software directly from the models. For these reasons, the quality of BPMs is critical for the success of software development. This raises the question, how we can benefit from well-established practices for improving the quality of software if we switch from code-centric to BPM-centric software engineering. In this article, we discuss how to apply concepts comparable to structured programming to BPMs. The main contribution is a discussion of the benefits of style checking for improving the quality of BPMs.

By analyzing 285 BPMs (modeled as Event Driven Process Chains (EPC)), we found that checking restrictions for "good modeling style" has three positive effects: It can improve the quality of the BPM by substituting "bad constructs" automatically, it helps to identify erroneous models and it can make model-to-code transformations much easier.

Keywords: Business process modeling; Modeling style; Style rules

1. Introduction

One of the major challenges in the field of enterprise information systems is the alignment of the software with the underlying business processes. Usually, the specialists who have deep knowledge about the business processes are not the same persons who develop the enterprise software. This implies that the business processes need to be discussed between domain experts and software developers. Graphical business process models (BPM) can support this communication. Working with such models has become a considerable part of developers' work. In contrast to old-style code-driven development, developers have to work extensively with BPMs before writing even a single line of code.

From the above points, it becomes clear that the quality of BPMs is essential for the success of enterprise software development. BPMs must not only be correct, but also easy to comprehend and easy to maintain. In this paper, we investigate how we can profit from existing research about code quality in order to improve the quality of BPMs.

* Corresponding author.
E-mail addresses: gruhn@ebus.informatik.uni-leipzig.de (V. Gruhn), laue@ebus.informatik.uni-leipzig.de (R. Laue).
[1] The Chair of Applied Telematics/e-Business is endowed by Deutsche Telekom AG.

0167-6423/$ - see front matter
doi:10.1016/j.scico.2006.08.003

In Section 2, we discuss the well-known concept of structured programming and some ideas on how this concept can be adopted for BPM. In the following sections, we pay special attention to coding style rules (also known as code conventions) and – as their counterpart – modeling style rules. In Section 3, we briefly introduce the BPM language EPC and its semantical problems that arise from the so-called non-locality of OR-joins. In Section 4, we define an example modeling style rule for EPC models. We introduce a rule for "good modeling style" and also transformation rules for correcting common errors automatically, as discussed in Section 5. Section 6 gives some quantitative information about how our style rule can help to improve the quality of "real-life models" that we have collected from several sources. These results can be found in Section 7. In Section 8, we discuss some related research areas and outline the directions of our future research.

As related work is discussed throughout this paper, there is no explicit "related work" section.

2. Structured programming and structured modeling

Nowadays, structured programming [1], a concept that emerged in the 1970s, is the standard practice for developing software. It is characterized by structured control blocks that can be assembled into larger software blocks. In particular, structured control mechanisms (like repeat-until) are used instead of the error-prone goto statement.

Today's BPM languages, on the other hand, often do not even have syntactic constructs for structured cycles. Holl and Valentin [2] write that "the current unstructured style of business process modeling, which we can call spaghetti business process modeling, leads to similar problems as spaghetti coding". Indeed, the problems are almost the same. In his famous article [3], Dijkstra states that programmers should do their "outmost to ... make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible". This statement remains true if we replace the term "program (spread out in text space)" by "graphical BPM".

It is our hypothesis that techniques similar to those developed in structured programming can help to improve the quality of BPMs. We expect that in the future, structured business process modeling will be as common as structured programming. Recent publications show that in most cases, unstructured models can be replaced by structured ones [4,5], and there are already BPM languages like BPEL4WS bpel4ws or the workflow management system ADEPT [6] with semantic restrictions which force the modeler to build well-structured models. With such languages, a modeler has no choice but to create well-structured BPMs. However, as long as the modeler has to use modeling languages that allow him to build unstructured models, he has still the possibility to restrict himself to use goto-like jumps in a way that doesn't obscure the flow of control in the BPM. In this paper, we discuss the hypothesis that such checks for "good modeling style" can improve the quality of BPMs.

Such style checks can be integrated into a BPM editor and used in a similar way as style checkers for software like Splint (www.splint.org), JLint (jlint.sourceforge.net) or FindBugs (findbugs.sourceforge.net) which are known for enhancing the code quality of software.

Before giving an example for style rules for the BPM language EPC (Event Driven Process Chains) in Section 4, we will introduce this language and highlight its main semantical problems.

3. EPC — an example for a BPM language

3.1. Informal semantics

Event Driven Process Chains (EPC) are a popular technique for business process modeling. Unfortunately, their initial authors did not define their precise semantics. Instead, the informal semantics are given roughly as follows:

EPCs consist of three kinds of elements: functions (activities which need to be executed, depicted as rounded boxes), events (pre- and postconditions before/after a function is executed, depicted as hexagons) and connectors (which can split or join the flow of control between the elements). Arcs between these elements represent the control flow. Each EPC has one or more start events which carry tokens (called process folders) when the EPC becomes enabled and "is executed". These tokens are propagated through an EPC. Events and functions have at most one incoming and at most one outgoing arc, and the tokens are simply propagated from the incoming arc to the outgoing arc.

Connectors are used to model parallel and alternative executions. There are six types of connector:

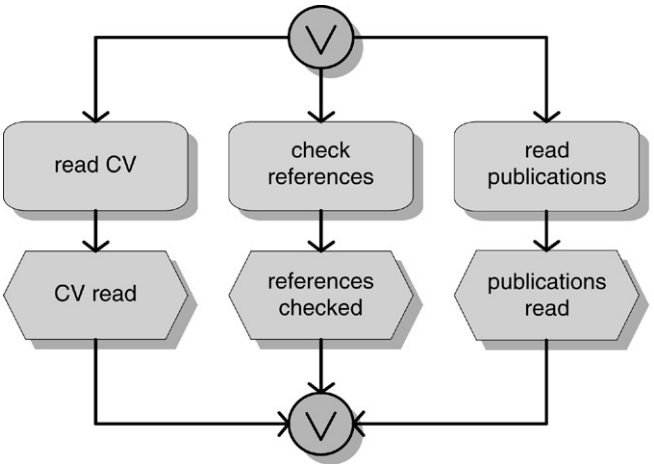

Fig. 1. Example EPC with OR-join.

AND-connectors (depicted as $(\wedge)$) are used to model parallel execution. The AND-split connector propagates a token from its incoming arc to all its outgoing arcs. The corresponding AND-join connector waits until a token has arrived on all incoming arcs before sending a token to the outgoing arc.

XOR-connectors (depicted as $(\times)$) can be used to model alternative execution: An XOR-split has multiple outgoing arcs, but an incoming token will be propagated to exactly one of them. The corresponding XOR-join waits for an incoming token on one of its incoming arcs and propagates it to the outgoing arc.

Finally, OR-connectors (depicted as $(\vee)$) are used to model parallel execution of one or more flows. An OR-split propagates an incoming token to one or more of its outgoing arcs. The corresponding OR-join waits until a token arrives on each of those incoming arcs that can deliver a token and propagates it to the outgoing arc.

Of course, the above informal description is insufficient and imprecise. However, the informal description is sufficient to interpret the meaning of the majority of EPC diagrams.

There are several proposals for a formalization of the semantics of EPCs: van der Aalst [7] proposes a mapping from EPCs without OR-connectors onto Petri nets. Langner, Schneider and Wehler [8] also translate EPCs for which certain well-formedness rules must hold onto Petri nets. Other papers on defining formal semantics include [9–11].

3.2. Problems with OR-joins

While the informal semantics discussed in Section 3.1 lead to a straightforward mapping of functions, events, split connectors, XOR- and AND-join connectors to Petri nets or other semantically well-founded models [7], there are serious problems with the OR-join connector.

The easiest (and most common) usage of this connector is depicted in Fig. 1, which shows a part of the process for recruiting and selecting academic staff. One, two or all branches after the OR-split may be processed (by sending tokens). The corresponding OR-join must wait until all tokens have arrived.

However, the decision whether more tokens can arrive on one of the incoming arcs cannot be made locally at the OR-join. As the OR-split can send tokens to one, two or all three outgoing arcs it is not even known whether the OR-join has to wait for one, two or three incoming tokens. As the "firing condition" cannot be checked locally at the OR-join, the semantics of this connector are called *non-local*. This non-locality leads to serious problems when the formal semantics of the OR-join have to be defined. A detailed discussion of these problems is beyond the scope of this paper. (We refer the reader who is interested in this topic to [7,8,12,13,9,14].) The non-locality of OR-joins can even raise problems to the effect that it is *impossible* to define a formal semantics of EPCs that is fully compliant with the informal semantics. Van der Aalst et al. [15] give a nasty example (called the vicious circle), an EPC with two OR-joins in a feedback loop, each of which waits for the other to complete first. For this EPC, it is not possible to define formal semantics in a satisfying way.[2]

[2] Some authors, for example Kindler [15,9,14], interpret the meaning of an XOR-join with non-local semantics as well. We follow the approach suggested by van der Aalst [7] instead and assume that an XOR-join has local semantics and forwards every token that arrives.

From a theoretical point of view, these problems have been solved: Kindler [9] uses techniques from fixed point theory in order to define the semantics for EPCs (taking into account that there are EPCs for which no suitable semantics exist [15]). Wynn et al. [13] and Cuntz et al. [14] use backward marking and state space exploration techniques for calculating the enabledness of an OR-join. Both approaches calculate the semantics of moderately sized models in a reasonable time. They aim to find the semantics for every EPC where suitable semantics exist. Obviously, this is the best possible solution from a theoretical point of view. We argue, however, that in practice, it is not really desirable to compute the semantics of *every* EPC model. If a BPM is modeled in a hard-to-read style, it is very unlikely that domain experts who use this model as a basis for communication will find it useful.

For this reason, we asked *for which kind of models* the semantics of the OR-join are difficult to define. As we will show in Section 4, the answer is that these problems occur for such models that are modeled in what we call "bad modeling style". Unfortunately, such models are not uncommon, because EPCs (and other business process modeling languages as well) do not require proper nesting, i.e. splits and joins do not have to occur pairwise. This is comparable with the situation in unstructured programming languages. To overcome these problems, we follow the ideas of structured programming. We wondered whether it would be possible to find a rule for "good modeling style" for OR-joins that:

(1) Does not "forbid" existing EPCs on whose correctness domain experts already agree.
(2) Does not seem to be "artificial" or "surprising". (Good modelers should already follow the rule intuitively.)
(3) Can be checked easily.
(4) And (as a result of the "structuredness") ensures that models that follow the rule can be transformed into semantically well-founded models like Petri nets using an easy-to-implement algorithm (in particular, this means that these models *have* well-defined semantics).

The next section describes a style rule for OR-joins meeting these requirements.

4. Style rule for OR-joins

By analyzing EPCs from several sources (see Section 6), we identified a suitable style rule for OR-joins. OR-joins that follow this rule can be transformed into formally founded languages (i.e. they have clear semantics), and our rule does not unnecessarily reject too many EPCs as being not well-formed.

The style rule can be summarized in the following recursive definition. (In this definition, we abstracted away from functions and events in the EPC, because the critical part is in the connectors.)

Definition 1 (*Well-structuredness; Style Rule for OR-joins*).

(1) The workflow constructs shown in Fig. 2 are well-structured. While Fig. 2(a)–(c) only shows connectors with exactly 2 outgoing/incoming arcs, it is also allowed that there are more than 2 arcs between the split connector and the join connector.
(2) If a well-formed construct is "inserted into an arc" of a well-structured construct (see Fig. 3), the resulting construct is well-structured.
(3) If an additional split is "inserted into an arc" of a well-structured construct (see Fig. 4, but note that the split does not have to be a XOR-split), the resulting construct is well-structured.
(4) If an arc of a well-structured construct is replaced by an event (to model termination of the flow of control at this point, see Fig. 5), the resulting construct is well-structured if the graph formed by the arcs and connectors is still a connected graph.

An EPC follows the *style rule for OR-joins* iff for every OR-join the construct between its corresponding OR-split and the OR-join is well-structured.

Roughly speaking, rules 1 and 2 ensure that each join-node corresponds to a split-node. Rule 3 allows "jumping out of a split-join construct", but it forbids "jumping into a split-join construct". The idea behind rule 3 is that the use of goto-like jumps should be restricted to situations that are comparable to exception handling in modern structured programming languages. Please note that it follows from Definition 1 that every join node has a corresponding split node of the same type, and in particular every OR-join must have a preceding OR-split.

For OR-joins that follow the style rule, we can define precise semantics using the idea of Langner, Schneider and Wehler: In [8], they use colored Petri nets with tokens marked with "0" (for "do not activate") and "1"

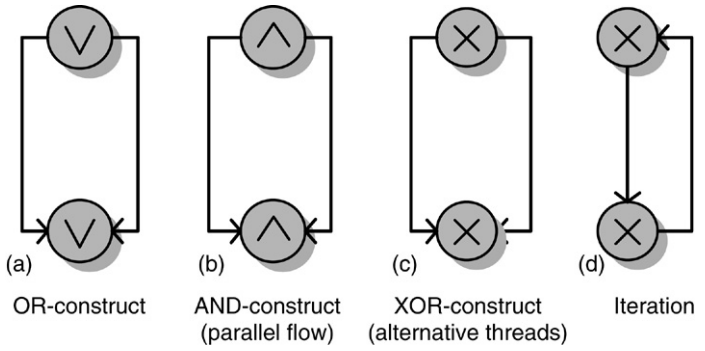

Fig. 2. Workflow constructs.

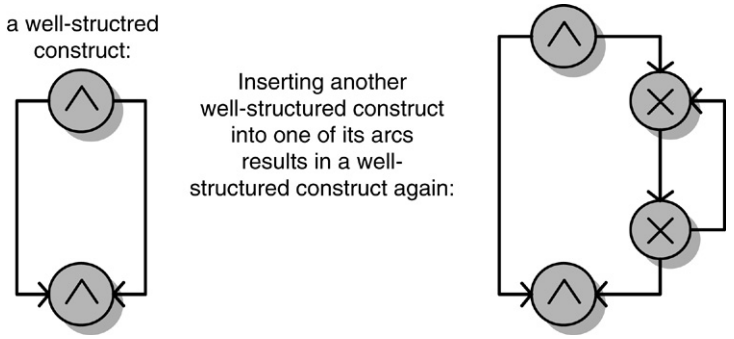

Fig. 3. Definition, part 2.

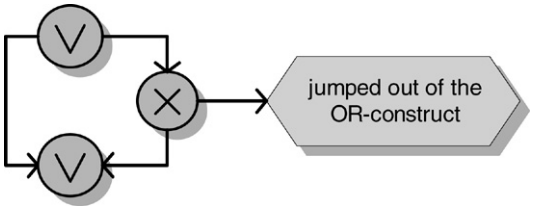

Fig. 4. Definition, part 3.

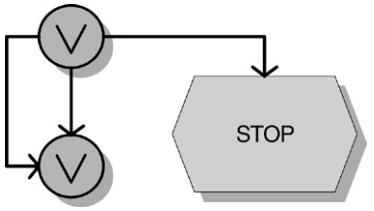

Fig. 5. Definition, part 4.

(for “activate”). An OR-split sends tokens along all outgoing paths: a “1” to activate the path or a “0” to deactivate it. The corresponding OR-join waits for the arrival of tokens from all incoming paths and it forwards a token if at least one incoming token is marked with “1”. A jump out of the split-join construct is not a problem, because a “0” token can be sent if the flow of control leaves the split-join construct. This means that an EPC in which all OR-joins are well-structured has clear semantics. While in some cases (see [4] for examples) it is also possible to define clear semantics for not well structured OR-joins (using approaches like [13,9] or [14]), such models should be avoided because they might be hard to understand by domain experts.

The style rule defined in Definition 1 can be checked statically (i.e. no simulation is needed).

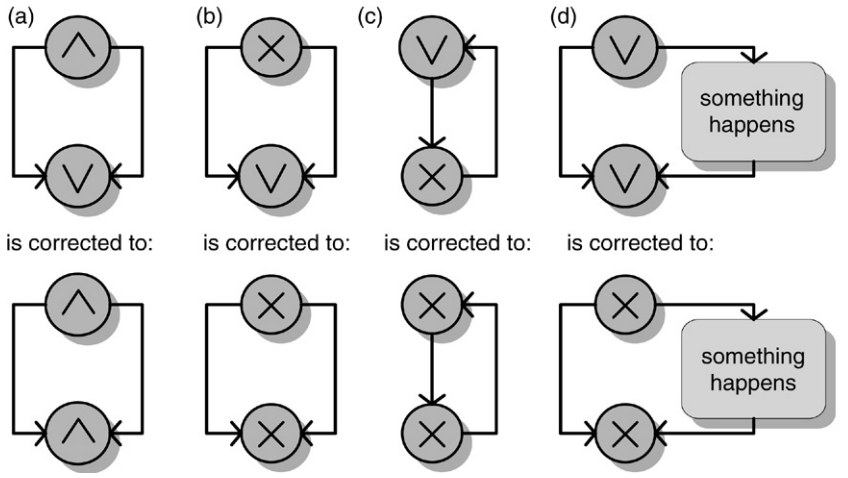

Fig. 6. Common errors and their corrections.

5. Correcting common errors automatically

If a model fails the test of the style rule, we can often use the results of its analysis for fixing it automatically. In our analysis of existing EPCs, we found a surprisingly large number of models with common errors that can (and should) be improved automatically based on the results of this static analysis. Fig. 6 shows these common errors and their correction.

In particular, we found that modelers often used OR-joins when an XOR-join or an AND-join would describe the desired behavior much better. (See Fig. 6(a)–(c), where we have omitted functions and events, because the error lies in the use of connectors only). Another common error was to use OR-split/joins instead of XOR-constructs in order to model optional execution (Fig. 6(d)).

While the "bad" constructs would be allowed according to the informal semantics, they should be changed into models with an AND/XOR-join anyway. Note that these corrections can change a construct that is not well-structured into a well-structured one *without* changing its (informal) semantics. However, even if the semantics of the model remain the same, the corrected one is more explicit and less likely to become the source of misunderstandings and faulty implementation.

Static analysis can also detect other errors that cannot be fixed automatically. While we restrict our focus to errors that are related to OR-joins in this paper, we note that similar style rules can be defined for finding other classes of errors (for example, constructs that start with an XOR-split but end with an AND-join which will result in a deadlock).

6. Survey

We have collected 285 EPCs from different sources (23 Masters theses, 2 term papers, 4 Ph.D. theses, 5 textbooks, 30 scientific papers, lecture notes from a university course, the SAP reference model [16] and one of our own projects — the full list of sources can be found on the web [17]). We tolerated EPCs with small syntactical errors, but nine graphics called an "EPC" had such significant syntactical problems that we would not regard them as actually being EPCs. We ignored them in the further analysis.

For the 276 remaining EPCs, we found that:

- 190 EPCs did not use OR-joins at all.
- The remaining 86 EPCs with OR-joins contained 151 OR-join connectors. 94 of these connectors were either used in the trivial way (with a corresponding OR-split and without any other connectors between the OR-split and the OR-join) or in a well-structured way according to Definition 1.

The most interesting result was the analysis of the remaining 57 OR-joins in not well-structured constructs: 45 of them fell into one of the cases depicted in Fig. 6, i.e. they should be replaced by another join node. For ten other not

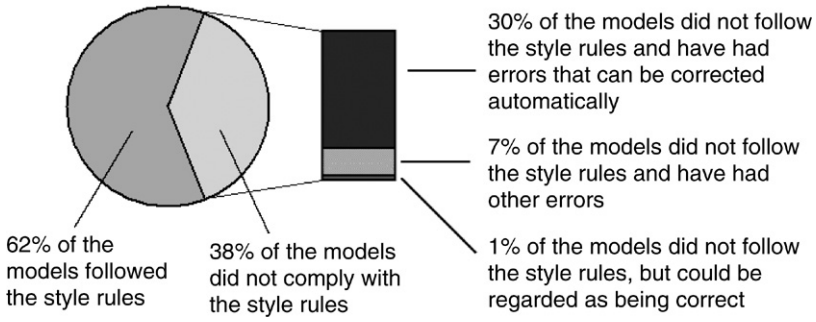

Fig. 7. Results of the survey.

well-structured models with OR-joins, a closer inspection revealed that the model was faulty,[3] and the error could not be corrected automatically. This indicates that syntactically correct models that do not obey our style rule are most likely erroneous, and applying the style rule can help to identify the errors almost immediately when the model has been drawn.

There were only 2 EPCs which could be regarded as being correct (i.e. the business process would come to the desired result), but failed to comply with the style rule from Definition 1. However, these EPCs were also modeled in an unsatisfactory way; in particular the soundness property as defined in [18] was violated for these models. Both models would profit from a re-design.

Fig. 7 shows the results of our survey in a diagram.

7. Discussion

The main result of the survey in Section 6 is that if the analysis found that an EPC did not comply with the style rule for OR-joins, in the vast majority of cases, this meant that in fact this model was erroneous. Interestingly, most of these errors could be corrected automatically as the result of the static analysis.

When used in an enterprise, our approach may be able to detect fewer errors because we suppose that in an enterprise environment, there are less faulty models than in our survey of models mainly from academic and tutorial sources. However, the fact that there are a considerable number of flaws even in models that are supposed to be developed by very well-experienced modelers is backed by [19]. In this survey, it has been found that at least 5.6% of EPC models in the SAP reference model [16] contain errors. Because the analysis in [19] can detect a certain class of errors only, this is only a lower bound for the number of errors. In general, we believe that there is a good chance that the use of style rules can improve the quality of BPMs in an enterprise.

We would like to stress that the fact that there were only two correct models that failed the style check is a result of our approach: We studied a large number of existing models *before* formulating the style rule for OR-joins. For this reason, the coverage of our rule for OR-joins is much better than in existing approaches, in particular [8]. The problem with the restrictions posed on EPCs in [8] is that they exclude too many existing EPCs from being regarded as well-formed: Too many models that are well-modeled and can be used successfully in practice would be regarded as being invalid.

While our style rule is less restrictive than the ones published in [8], we can still use the idea from [8] to transform EPC models that follow the style rule into a Boolean net, a Petri net variant (formally speaking, this transformation *defines* the semantics of the EPC). The translation into a Boolean net is a generalization from the algorithm published in [8]; details can be found in [20].

Our survey deals with the question of how errors in *existing* BPMs can be found and corrected by applying style rules. An interesting point for future research is how errors can be *prevented* if the use of these rules is enforced when the models are being built. (This is what [21] calls “verification by design”.)

The usage of style rules for preventing (or detecting) possible errors in a BPM is somewhat heuristic compared with other formal methods. However, this heuristic approach allows us to identify problems that are not detected by

[3] “Faulty” means here that the model either had structural defects like deadlocks or contained errors in the business logic. Problems of the latter kind can only be identified manually, since they require an understanding of the business process.

other methods. In particular, this was the case for the models that did not comply with the style rule but could not be fixed automatically. Most of them had an error in the business logic. An example from one of our own projects was an electric meter that was modeled as being working and being out of order at the same time. Finding such flaws requires the understanding of the underlying business process. Existing tools that check technical properties of a model (like absence of deadlocks etc.) are unable to find such errors. While human action is required to find the errors in the business logic, checking style rules can guide us to *which* models should be inspected manually because they most likely contain errors.

A good overview about current work on BPM verification can be found in [21]. Most of these approaches have in common that a BPM is mapped onto a formal model with executable semantics. Using model-checking techniques, it is possible to search for errors in these formal models. Checking style rules is no substitution for these approaches. The main difference between using a style-checker and using a model-checker is that the latter aims to find errors directly while the style-checker aims to find elements in a BPM that most likely lead to an error or at least can become the source of misunderstandings. So far, only little research has been dedicated to the question of which model elements lead to errors. The most elaborate research paper on this question is [19]. It combines formal error identification in EPCs with quantitative analysis of potential error determinants. This paper deals with the impact of a number of rather simple measures (like "number of connectors in an EPC") on the error-rate. It should be promising to use this approach for identifying the relationship between more complex "style elements" and the error rate, as suggested by the authors of [19] as a field for their future research.

We emphasize that the proposed approach of applying additional well-formedness checks to business process models is not restricted to EPCs. Other languages will require other style checking rules. For example, the syntax for UML Activity Diagrams allows syntactically correct but absolutely useless constructions [22] which should be forbidden by corresponding style rules.

The style rule given in Definition 1 deals with one particular problem (the OR-join semantics) only. This is just one aspect of structured modeling, and of course, there are many more potential problems in graphical BPMs that can be regarded as "bad style". Currently, there is not much published work about this subject. For EPC models, [8] lists some defects that can be detected using style rules. For UML diagrams, several useful rules can be found in [23,24].

8. Conclusion and future work

We found that requiring and checking additional restrictions for "good modeling", as described in Section 4, has three positive effects:

(1) The algorithm for translating the model into another (formally founded) model that can be used in model-checkers or simulation tools becomes easier (compared with algorithms like the ones from [25] or [14]).
(2) The model can be improved automatically by correcting common errors (see Section 5).
(3) Failing the style checks most likely means that the model is erroneous. This fact helps us to detect errors immediately after the model has been drawn (and correct them manually).

Because of these encouraging results, we will continue our research on style rules. We will also try to use experiences gained from research on software quality for improving BPM quality as outlined in the following subsections.

8.1. Layout and textual quality

A basic statement in [26], the fundamental book on coding style rules, is that code should be written not only for the compiler, but mainly for a human reader. The rules discussed in this book include "Format a program to help the reader understand it"! and "Choose variable names that won't be confused"! We are convinced that it is even more important to consider similar rules for graphical BPMs whose purpose is to support the discussion between domain experts and developers. In a graphical BPM that should be understood by a human reader, attention must be paid to both the layout of the graphics and the quality of the textual descriptions. While there are already useful collections of layout and style rules for graphical models (such as [24] or [27]), a considerable number of business process modelers still do not use them. We think that there is still much room for improving the layout and textual quality of BPMs. From an academic point of view, empirical studies on how BPM comprehension is affected by layout and style should be a fruitful area for future research.

8.2. Metrics for complexity and comprehensibility

In this article, we have often used phrases like "a BPM that is easy to understand", but we have not yet given formal definitions for them. It would be useful to have measures that can give us some information about understandability and complexity of a BPM. Such measures should tell us whether the model has an appropriate size, is clearly structured, easy to comprehend and partitioned into modules in a sensible way. Once again, we can profit from the experiences on software. Research results on complexity metrics for software can be extended in order to analyze the complexity of BPMs. Some suggestions related to this field of research can be found in [28,29,19], where only the latter contains a quantitative analysis of how good the metrics work for predicting errors. Because such a validation of complexity metrics is necessary before using them in practice, it will be a subject of our future research.

8.3. Patterns

Software design patterns [30] are a successful method for improving the quality of software and reducing its complexity. Design patterns are well-documented solutions to recurring problems that have been found highly mature. For BPM, reference models [16] can be seen as one form of design pattern. They have been successfully used for (re)designing, tailoring, and implementing business processes. While these reference models have been found to be valuable as templates for high-level business processes, it would be useful to have a pattern catalog for "fine-grained" recurring situations in business process modeling (e.g. "enforcing four-eye-principle") as well. Several proprietary workflow management systems offer support for a selection of such patterns, but there is less work on building a general-purpose pattern catalog [31–33].

We are convinced that the ideas presented in this article can help to improve the quality of BPMs. While structuredness, style and complexity of software are well-studied research areas, the corresponding questions about structuredness, style and complexity of BPMs still leave considerable room for future research. We will focus our research on measuring and reducing complexity of BPMs and measuring the impact of style and complexity on the quality of a model.

References

[1] E.W. Dijkstra, Notes on Structured Programming, Academic Press, London, UK, 1969.
[2] A. Holl, G. Valentin, Structured business process modeling (SBPM), in: Information Systems Research in Scandinavia (IRIS 27) (CD-ROM), 2004.
[3] E.W. Dijkstra, Letters to the editor: go to statement considered harmful, Communications of the ACM 11 (3) (1968) 147–148.
[4] B. Kiepuszewski, A.H.M. ter Hofstede, C. Bussler, On structured workflow modelling, in: Conference on Advanced Information Systems Engineering, 2000, pp. 431–445.
[5] R. Liu, A. Kumar, An analysis and taxonomy of unstructured workflows, in: Business Process Management, 2005, pp. 268–284.
[6] M. Reichert, P. Dadam, ADEPTflex-supporting dynamic changes of workflows without losing control, Journal of Intelligent Information Systems 10 (2) (1998) 93–129.
[7] W. van der Aalst, Formalization and verification of event-driven process chains, Information & Software Technology 41 (10) (1999) 639–650.
[8] P. Langner, C. Schneider, J. Wehler, Relating event-driven process chains to Boolean Petri nets, Report (9707).
[9] E. Kindler, On the semantics of EPCs: A framework for resolving the vicious circle, in: Business Process Management, 2004, pp. 82–97.
[10] J. Dehnert, W. van der Aalst, Bridging the gap between business models and workflow specifications, International Journal of Cooperative Information Systems 13 (3) (2004) 289–332.
[11] B.F. van Dongen, W. van der Aalst, H.M.W. Verbeek, Verification of EPCs: Using reduction rules and Petri nets, in: CAiSE, 2005, pp. 372–386.
[12] P. Rittgen, Quo vadis EPK in ARIS? Wirtschaftsinformatik 42 (1) (2000) 27–35.
[13] M.T. Wynn, D. Edmond, W. van der Aalst, A.H.M. ter Hofstede, Achieving a general, formal and decidable approach to the OR-Join in workflow using reset nets, in: ICATPN, 2005, pp. 423–443.
[14] N. Cuntz, J. Freiheit, E. Kindler, On the semantics of EPCs: Faster calculation for EPCs with small state spaces, in: EPK 2005, Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, 2005, pp. 7–23.
[15] W. van der Aalst, J. Desel, E. Kindler, On the semantics of EPCs: A vicious circle, in: EPK 2004: Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, 2002, pp. 71–79.
[16] A.-W. Scheer, Business Process Engineering: Reference Models for Industrial Enterprises, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1994.
[17] R. Laue, ebus.informatik.uni-leipzig.de/~laue, 2005.
[18] W. van der Aalst, The application of Petri Nets to workflow management, The Journal of Circuits, Systems and Computers 8 (1) (1998) 21–66. URL citeseer.ist.psu.edu/vanderaalst98application.html.

[19] J. Mendling, M. Moser, G. Neumann, H. Verbeek, B. Dongen, W. van der Aalst, A quantitative analysis of faulty EPCs in the SAP reference model, Tech. Rep. BPM-06-08, BPM Center Report, BPMcenter.org, 2006.
[20] V. Gruhn, R. Laue, Einfache EPK-Semantik durch praxistaugliche Stilregeln, in: Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, 2005, pp. 176–189.
[21] B.F. van Dongen, M.H. Jansen-Vullers, EPC verification in the ARIS for MySAP reference model database, in: BETA Working Paper Series, WP 142, Eindhoven University of Technology, Eindhoven, 2005.
[22] H. Störrle, Semantics of UML 2.0 activities, in: Symposium on Visual Languages — Human Centric Computing (VL/HCC'04, Proceedings), IEEE, 2004, pp. 235–242.
[23] R. Gronback, Model validation: Applying audits and metrics to UML models, in: Borland Conference, 11–15 September 2004, San Jose, California, USA, 2004.
[24] S.W. Ambler, The Elements of UML Style, Cambridge University Press, 2003.
[25] N. Cuntz, E. Kindler, On the semantics of EPCs: Efficient calculation and simulation, in: EPK 2004: Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, Proceedings, 2004, pp. 7–26.
[26] B.W. Kernighan, P.J. Plauger, The Elements of Programming Style, McGraw-Hill, Inc., New York, NY, USA, 1982.
[27] H. Koning, C. Dormann, H. van Vliet, Practical guidelines for the readability of it-architecture diagrams, in: SIGDOC '02: Proceedings of the 20th Annual International Conference on Computer Documentation, ACM Press, New York, NY, USA, 2002, pp. 90–99.
[28] J. Cardoso, How to measure the control-flow complexity of web processes and workflows, in: The Workflow Handbook, 2005, pp. 199–212.
[29] V. Gruhn, R. Laue, Complexity metrics for business process models, in: 9th International Conference on Business Information Systems, BIS 2006, Klagenfurt, Austria, 2006.
[30] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: Abstraction and reuse of object-oriented design, Lecture Notes in Computer Science 707 (1993) 406–431.
[31] P. Rittgen, K. Turowski, UML design patterns for business processes, in: International Conference of the International Resources Management Association, IRMA, 2002, pp. 679–681.
[32] R. Motschnig-Pitrik, P. Randa, G. Vinek, Specifying and analysing static and dynamic patterns of administrative processes, in: Xth European Conference on Information Systems, ECIS, 2002.
[33] L.H. Thom, C. Iochpe, B. Mitschang, Improving workflow project quality via business process patterns based on organizational structure aspects, in: XML4BPM 2005, Proceedings of the 2nd GI Workshop XML4BPM — XML Interchange Formats for Business Process Management at 11th GI Conference BTW 2005, Karlsruhe, Germany, 2005, pp. 65–80.